Fabule: Un environnement de recherche pour l'animation et la simulation

Jean-Dominique Gascuel

Fabule : Un environnement de recherche pour l'animation et la simulation.

Jean-Dominique Gascuel
CNRS EP J0038, iMAGIS/IMAG * – INRIA
Jean-Dominique.Gascuel@imag.fr

Résumé

La plate-forme de développement *Fabule* a été mise au point pour répondre aux besoins communs du groupe animation d'iMAGIS. Cet environnement nous permet de factoriser les efforts de programmation de chacun dans un cadre cohérent et, autant que possible, complet.

Le but premier du système est de réduire les délais entre la conceptualisation d'un algorithme nouveau, les tests et l'expérimentation dans un cadre interactif, et pour finir la production d'une séquence vidéo illustrant les résultats. Pour que plusieurs chercheurs puissent expérimenter de nouveaux modèles (il s'agit principalement d'animation par "modèles physiques") dans le cadre de *Fabule* nous avons opté pour une bibliothèque logicielle orientée objet, interfacée avec un langage interprété de manière à faciliter la génération de démonstrateurs.

Cette présentation, qui détaille l'architecture logicielle que nous avons mise en place, s'articule autour des points suivants : buts et architecture globale du système, primitives graphiques, et modules d'animation.

1 Buts et Architecture

Fabule est un environnement de développement dédié à la **recherche en animation**, au sein d'une équipe qui s'intéresse plus particulièrement à la mise au point de "modèles physiques" pour l'animation et la simulation. Les principales considérations qui ont accompagnées la conception de *Fabule* étaient d'avoir un système qui permette à un groupe de chercheurs d'implémenter de nouveaux modèles et algorithmes pour l'animation, de les tester dans des conditions variées, et de produire rapidement des séquences vidéo montrant aussi bien des cas d'école que des situations complexes.

Plus précisément, les points suivants ont guidés la conception du système :

Gain de temps : L'environnement doit permettre de raccourcir le temps passé entre les premiers essais d'un algorithme nouveau, la visualisation de résultats, et la production d'un film de démonstration. L'environnement doit donc inclure des outils permettant d'aborder les différentes étapes, depuis la modélisation, l'animation, la production d'images (et de son), jusqu'à la post-production.

∗. L'institut d'Informatique et de Mathématiques Appliquées de Grenoble est une fédération de laboratoires CNRS, INPG et UJF.

Programmation : Puisque qu'il s'agit de permettre aux chercheurs d'expérimenter des approches et des modèles encore inexplorés, *Fabule* ne peut être conçu comme une application finie, fermée et bien rodée. En revanche, notre système propose un ensemble cohérent de concepts, dans lesquels chacun peut puiser ce dont il a besoin pour le développement de sa démonstration.

Ré-utilisation : Pour que le système grandisse et permette d'utiliser les résultats acquis dans d'autre recherches, nous avons décidé d'imposer une structuration modulaire. Cela s'est traduit par une programmation orienté objet (**C++**), l'utilisation d'interfaces abstraites, de méthodes virtuelles, et le choix d'un langage interprété (***Tcl***) pour la programmation des interfaces.

Ne pas ré-inventer la roue : Enfin, il s'agissait de minimiser les efforts de programmation et de maintenance. Pour ce faire, *Fabule* repose sur un maximum d'outils commerciaux ou domaine publique, dont la maintenance ne repose pas sur nous...

Architecture

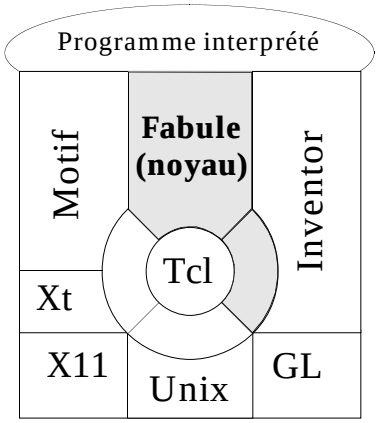

FIG. 1 - Fabule *: les différentes bibliothèques de l'interpréteur. Le langage tient la place centrale, entouré par les interfaces graphiques (2D à gauche, 3D à droite). En grisé, la bibliothèque d'animation et les interfaces que nous avons développées.*

Ces buts ont été concilié dans *Fabule* par l'utilisation de l'architecture suivante (Figure 2) :

- Un interpréteur — basé sur le langage *Tcl* — permet de programmer les interfaces, les données graphiques, et les objets d'animation propres au cœur de *Fabule*(figure 1).

  Les interfaces sont programmées grâce à la bibliothèque *OSF/Motif*. Les données graphiques sont manipulées et affichées grâce à la base de données interactive *OpenInventor*.

  Dans ce langage sont écrit : les programmes d'animation, les modeleurs spécifiques (objets implicites), et les outils d'aide à l'animation (choix des lampes, de la caméra, ...)

  Les données graphiques sont stockés suivant le format de fichier d'*OpenInventor*.

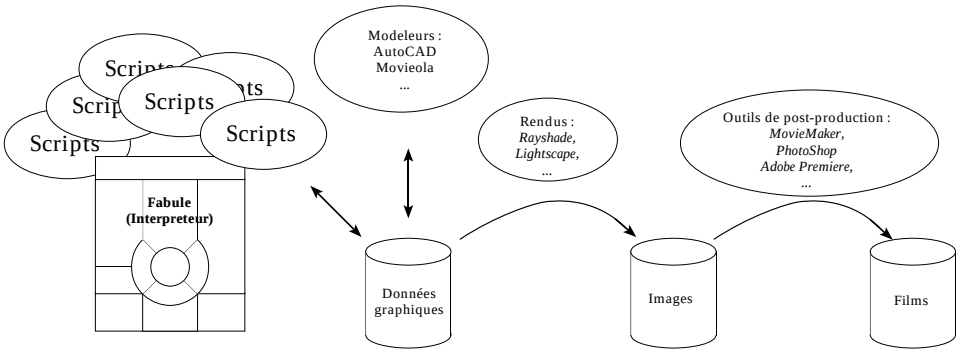

FIG. 2 - *Fabule : de la modélisation à la vidéo de démonstration. A gauche, l'interpréteur avec différentes applications, écrites en* Tcl. *Viennent ensuite les différents outils formant l'environnement de production d'images ou de films.*

- Des modeleurs compatibles *OpenInventor*: *AutoCAD*, *Movieola*, ...) permettent de créer des bases de données graphiques pour les objets non implicites, des éléments de décor, ...
- Des programmes de rendu permettent de créer des images hautes résolution à partir de base de données *OpenInventor* (nous utilisons en particulier le logiciel domaine publique *RayShade*).
- Enfin, ces images sont assemblées sous formes de séquences, qui sont montées en utilisant une plateforme *Macintosh* (*Adobe PhotoShop*, *Adobe Premiere*).

2 Primitives graphiques

Nous avons choisi d'écrire *Fabule* au dessus de la base de donnée graphique interactive *OpenInventor*, écrite en *C++*. Cette bibliothèque, basé sur *OpenGL*, a été introduite par *Silicon Graphics* dans le même consortium, et doit donc permettre une large portabilité du logiciel (*DEC*, *IBM*, *Microsoft*, et aussi *Sun*, *HP*, *Macintosh*...).

Nous avons écrit une interface *Tcl* pour cette bibliothèque, ce qui permet d'avoir un langage interprété puissant, qui peut manipuler la base de données graphique (voir les exemples dans le reste de l'article).

La base de données *OpenInventor*

La base de données est constitué de nœuds, que l'on organise en arbres ou en graphes orientés acycliques (c'est a dire que l'on peut partager des sous-arbres).

Les nœuds terminaux (feuilles) permettent de spécifier :

- **des données :** coordonnées, normales, textures, ...
- **des attributs :** couleurs, modèle de lumière, transformation, type d'interpolation, ...
- **des primitives géométriques :** cubes, sphères, liste de triangles, splines, NURBS avec des découpes, ...

L'affichage, pris en charge par *OpenInventor*, s'effectue lors d'un parcours en profondeur d'abord du graphe acyclique. Les nœuds internes permettent de choisir

quand et comment les sous-graphes sont à prendre en compte (`Group`, `MultipleCopy`, `Switch`, `LevelOfDetail`, ...).

L'exemple ci-dessous (écrit en *Tcl*) définit une procédure qui construit un papillon multicolore, et retourne le pointeur vers le sommet du graphe correspondant :

```
proc butterfly {} {
    set fly [SoSeparator]
# Une liste de sommets
    $fly addChild [SoCoordinate3 {
      point [0 0 0, -.2   1 .1,  -1  1.2 .5, -1.2  .2 .6, -.8 -.3 .4,
             0 0 0, -.7 -.2 .08, -1  -.6 .1, -.7  -.8 .1, -.1 -1  0,
             0 0 0,  .1  -1  0,   .7 -.8 .1,  1   -.6 .1,  .7 -.2 .08,
             0 0 0, .8  -.3 .4,  1.2 -.2 .6,  1   1.2 .5,  .2  1  .1]
    }]
# Une couleur pour chacun (modulo 5)
    $fly addChild [SoMaterial {
    diffuseColor [0 0 0, 0.2 0 1, 0.06 0.6 1, 0 1 0.4, 1 0.9 0.0]
    }]
# et on construit un polygone (qui interpole les couleurs) :
    $fly addChild [SoMaterialBinding {value PER_VERTEX}]
    $fly addChild [SoFaceSet]
    return $fly
}
```

Interactivité

En plus d'une représentation efficace et très complète de données, *OpenInventor* propose aussi des outils pour créer des interfaces interactives :

- **Des fenêtres** ("Widgets" *X-toolkit*) permettant de visualiser un sous-graphe, avec la possibilité de choisir le point de vue à la souris, et de sélectionner des objets.

  L'exemple suivant (toujours en *Tcl*) définit une procédure qui ouvre une nouvelle fenêtre *OSF/Motif*, et affiche dedans une scène *OpenInventor* quelconque. L'utilisateur peut alors interagir avec la souris pour changer le point de vue, le zoom, etc.

  ```
  proc view {scene} {
  # Creation d'une fenetre Inventor independante :
      topLevelShell .view
      SoXtExaminerViewer .view.viewer
  # Initialisations (la scene a afficher, le zoom initial, ...)
      .view.viewer setSceneGraph $scene
      .view.viewer viewAll
      .view.viewer saveHomePosition
      .view.viewer show
      .view popup
  }
  ```

- **Des éditeurs spécialisés** pour choisir une couleur, un matériau, ou les paramètres d'une transformation.

  La procédure suivante recherche un nœud de définition de matériau, et s'il le trouve ouvre une nouvelle fenêtre *OSF/Motif*, avec le dialogue standard

d'édition :

```
proc editMaterial {node index} {
    set mat [searchType $node SoMaterial]

    if {$mat != {}} {
        topLevelShell .sh
        SoXtMaterialEditor .sh.edit
        .sh.edit attach $mat $index
        .sh.edit show
        .sh popup
    }
}
```

- **Des "dragger" et des manipulateurs** pour interagir à la souris avec la base de données (sélectionner, déplacer, déformer, ...). Dans l'exemple suivant, on utilise un tel "widget 3D" pour déplacer interactivement un point d'un nœud de coordonnées :

```
proc dragPoint {scene index} {
    global root
# On recherche un noeud de type "table de sommets"
    set coord [searchType $scene SoCoordinate3]
    if {$coord == {}} {error "No coordinate node found"}
# Si on le trouve, on y attache un ``dragger'',
    set drag [SoDragPointDragger]
    $drag translation [$coord point($index)]
# dont le deplacement deplace le dit sommet,
    $drag addMotionCallback "setPoint $drag $coord $index"
# et qui disparaitra tout seul.
    $drag addFinishCallback "$root removeChild $drag"

    $root addChild $drag
}

proc setPoint {dragger coord index} {
    $dragger point($index) [$dragger translation]
}
```

3 Animation

A partir de cette base de données, une animation interactive peut être réalisée simplement, par mises à jours successives des nœuds de coordonnée et de transformation du graphe décrivant la scène.

Mouvements de base

Pour produire un mouvement, *Fabule* propose la notion de **générateur**. Ces entités génèrent le mouvement d'un repère local au cours du temps. soit par interpolation entre positions clés (animation descriptive), soit en intégrant les équations du mouvement d'un modèle (animation par "modèle physiques").

Fabule propose les générateurs de base suivants :

`KeyFrame` L'animateur spécifie un jeu de positions (translations et orientations), qui sont interpolées par des splines de Hermite (en utilisant les logarithmes de quaternions pour les orientations). Les tangentes peuvent être calculées automatiquement, avec des coefficients simplifiés (Tension, Continuité et Biais), ou manipulées directement (voir le rapport de DEA [1]).

`PointDynamic` L'intégration des forces (continues ou instantanées) et l'utilisation de la dynamique du point permet de générer des mouvement réalistes (les objets bougent suivant leur poids, leur inertie, ...).

`Dynamic` L'introduction de couples et de tenseurs d'inertie permet de simuler les rotations, en utilisant la mécanique du solide.

D'autre part, il y a plusieurs manières d'agir sur un générateur pour modifier sont mouvement. L'utilisation de **forces**, par une intégration du second ordre, permet d'engendrer des "mouvements lisses".

Dans des situations particulières (par exemple la simulation de choc rigides entre objets), on désire avoir des changements brusques de vitesse. Les forces ne donnent pas de bon résultats dans ces conditions ; puisque le caractère instantanné de la réponse implique l'utilisation de forces très grandes, donc peu précises et difficilement intégrables. L'utilisation d'**impulsions**, grandeur du même ordre de grandeur que les vitesses, et demandant une intégration du premier ordre seulement. Ce qui permet d'engendrer des mouvement continus, mais qui changent brusquement de direction ou de vitesse.

Enfin les contraintes géométriques que nous utilisons, par exemple pour maintenir des articulations, doivent imposer des ajustement dans les positions relatives des générateurs les uns par rapport aux autres. Plutôt que de passer par des dérivations suivies d'une intégration, les "contraintes de déplacement" [2] proposent d'ajuster directement la position des générateurs.

Mouvements complexes

Fabule permet de simuler des objets complexes en reliant plusieurs générateurs. Deux notions sont utilisées :

- Une **liaison** permet de modéliser une intéraction spécifique entre deux générateurs.

  On retrouve dans ce groupe des ressorts (ou des contrôleurs) qui produisent des forces, des chocs qui produisent des impulsions, et des contraintes qui produisent des déplacements.

- Dans les **systèmes de particules**, au contraire, chaque générateur calcule ses interactions avec les autres générateurs à partir de ses propres paramètres. Les interactions peuvent être globales (particules à grand rayon d'intéraction), ou réduites à quelques voisins (topologie fixée).

La boucle d'animation

L'interpréteur *Fabule* utilise *OSF/Motif*, et sa boucle de traitement des événements. Pour simuler un pas de temps, il faut donc produire les événements correspondants. *Fabule* utilise la classe *OpenInventor* `SoTimerSensor` qui se base sur ce mécanisme pour déclencher l'exécution de la boucle de simulation principale, résumée sur la figure 3.

Les différents modules de la simulation s'enchaînent. Ce schéma de résolution utilise un interval de temps adaptatif : si l'un quelconque des modules détecte que le

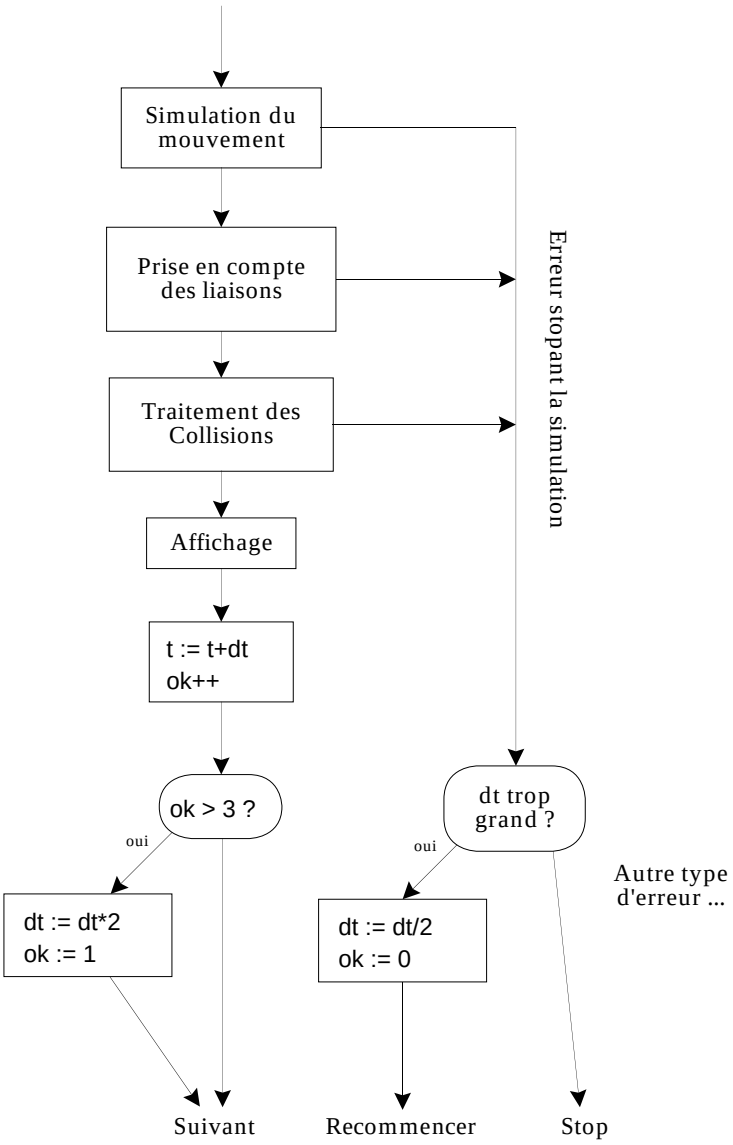

FIG. 3 - *La "boucle" d'animation : le traitement d'un pas de simulation, qui s'insere dans la boucle de traitement des évènements* OSF/Motif.

pas de simulation (dt) et trop petit pour rester dans des marges d'erreur acceptable, il renvois une erreur "pas de temps trop grand". On divise alors dt par deux, et on recommence la simulation. Si une autre erreur se produit, on stoppe la simulation : l'animateur peut alors examiner ce qu'il se passe et/ou relancer la simulation. Si la simulation termine sans erreur, et ce plusieurs fois de suite, alors on augmente dt (jusqu'à un certain maximum, typiquement $1/25^{\text{ème}}$ de second, pour obtenir les images d'un film d'animation), de façon à revenir automatiquement à un intervale de simulation aussi grand que possible.

4 Création de démonstrations

Comme indiqué au début, le but de *Fabule* est non seulement de permettre aux chercheurs de faire leurs recherches, mais aussi de montrer leurs résultats.

Interface Utilisateur

La première approche consiste à écrire un (petit) démonstrateur, qui peut d'ailleurs servir aussi à la mise au point des algorithmes. Il faut alors avoir la possibilité de décrire rapidement une interface graphique. Là encore, l'approche langage interprété (*Tcl*) permet de prototyper rapidement des dialogues, menus, etc.

L'exemple suivant permet de tester l'animation par position clef sur un cas d'ecole. Quelques lignes de *Tcl* suffisent à definir la trajectoire, le mobile, et a appeller l'interface :

```
proc makeScene {} {
# Creation du chemin par position clefs
    set p [FaPath]
    $p insertKey 0 [FaKey "0  0 0   0 0 0 1"]
    $p insertKey 1 [FaKey "1 -1 0   0 0 .707 .707"]
    $p insertKey 2 [FaKey "2  0 0   0 0 1 0"]
    $p insertKey 3 [FaKey "1  1 0   0 0 .707 -.707"]
    $p insertKey 4 [FaKey "0  0 0   0 0 0 1"]
# utilisation dans un generateur cyclic,
    set g [FaKeyFrame]
    $g setPath $p
    $g setCyclicTime 4.0 0.0
# qui anime un petit cube.
    set solid [FaRigidSolid]
    $solid setGenerator $g
    set solid_gr [SoSeparator]
    $solid_gr addChild [SoCube {width .4 height .4 depth .4}]
    $solid setPrimitive $solid_gr
# Calcul d'un polygone montrant le chemin (avec
# 10 points entre chaque position clefs),
    set coord [SoCoordinate3]
    $p computeDrawing $coord 10
    set line [SoSeparator]
    $line addChild $coord
    $line addChild [SoLineSet]
# que l'on met dans le graphe a afficher.
    global root
    $root addChild $line

    return $solid_gr
```

```
}

# Procedure predefini ouvrant une fenetre standard,
# et montrant la graphe construit par "makeScene"
simpleDemo
```

Séquences Vidéo et montage

On doit aussi pouvoir créer des séquences d'images à partir d'une animation interactive. Trois types d'approches sont possible. La première consiste à enregistrer directement la vidéo à partir d'une application interactive sur la station de travail. L'équipement nécessaire pour prendre les séquences est minimum (juste la station et le magnétoscope), mais sans studio vidéo, il est difficile de faire un montage complet, de bonne qualité.

La deuxième approche consiste à utiliser les technologies émergeantes nées du croisement du multi-média et de la vidéo numérique. L'idée consiste à produire une séquence d'image (par exemple *QuickTime*) sur la station de travail, et à faire le montage avec un logiciel du type *Adobe Premiere* sur un Macintosh. Depuis l'été 1994, on trouve des cartes graphiques et des disques rapides qui permettent de "jouer" le montage obtenu en pleine résolution, voire en qualité broadcast.

Dans *Fabule*, on peut calculer les images en utilisant *OpenInventor*, avec éventuellement un modèle, un décor, un environnement et une résolution différents de ceux utilisés en mode interactif.

On peut aussi sauver des fichiers pour des programmes de rendu externes. Ainsi, la totalité des films produits dans l'équipe ont été réalisés avec le logiciel de lancer de rayons *RayShade*.

5 Fabule aujourd'hui

Pour conclure, quelques chiffres sur le noyau d'animation de *Fabule*. Le développement de la version actuelle a débuté début 1994, et comporte actuellement 16.000 lignes de C++, et 1.500 lignes de *Tcl*.

Différents modules de recherche sont venus s'intégrer à ce noyau : des systèmes de particules (orientées ou non) (1500 lignes de C++)[3], la modélisation et l'animation d'objets implicites (5.000 lignes de C++ et 2.000 de *Tcl*)[4, 5, 6], et des méthodes de contrôle dynamique de trajectoires (1.500 lignes)[7, 8].

Références

[1] Ch. Lyon. Introduction de l'animation descriptive dans un contexte d'animation sous contraintes. *Rapport de stage*, DEA d'Informatique de l'ENSIMAG/UJF, September 1994.

[2] M.P. Gascuel and J.D. Gascuel. Displacement constraints for interactive modeling and animation of articulated structures. *The Visual Computer*, 10(4), March 1994. An early version of this paper appeared in the *Third Eurographics Workshop on Animation and Simulation*, Cambridge, UK, Sept 92.

[3] M. Desbrun. Modèlisation de matériaux hautement déformable en synthèse d'images. *Rapport de stage*, DEA d'Informatique de l'ENSIMAG/UJF, June 1994.

[4] Marie-Paule Gascuel. An implicit formulation for precise contact modeling between flexible solids. *Computer Graphics*, pages 313–320, August 1993. Proceedings of SIGGRAPH'93 (Anaheim, California, August 1993).

[5] Mathieu Desbrun and Marie-Paule Gascuel. Highly deformable material for animation and collision processing. In *5th Eurographics Workshop on Animation and Simulation*, Oslo, Norway, September 1994.

[6] N. Tsingos. Modèlisation interactive d'objets définis par des surfaces implicites. *Rapport de stage*, Magistère d'Informatique de l'Université Joseph Fourier, Grenoble, September 1994.

[7] Alexis Lamouret and Marie-Paule Gascuel. An approach for guiding colliding physically-based models. In *Fourth Eurographics Animation and Simulation Workshop*, pages 209–219, September 1993.

[8] Alexis Lamouret, Marie-Paule Gascuel, and Jean-Dominique Gascuel. Combining physically-based simulation of colliding objects with trajectory control. *A paraitre dans The Journal of Visualization and Computer Animation*, 1994.